CS 441 Programming Languages
Assignment #7

Due: Friday, 12 December 2008 (by midnight)

Preamble

In this assignment, there is a programming part and a theory part. You may do the programming part either individually or in pairs. If you work in pairs, each member of the pair will receive the same grade for the program. When you email your program files to Rob (rdockins@cs.princeton.edu), you should include in the email the names of all who contributed to the program.

Each person must write up and hand in the answers to the theory questions individually. You may toss around ideas with your classmates about the problems but you need to write up the end results yourself.

You have two weeks to complete the assignment. (Try not to leave it all to the very end!)

Type Soundness for Existential and Universal Polymorphism

Consider the lambda calculus with universal and existential types as we studied in class:

$$\begin{array}{lll} \text{Types} & \tau & ::= \ \texttt{int} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\tau \mid \exists \alpha.\tau \\ \text{Values} & v & ::= \ n \mid \lambda x{:}\tau.e \mid (v_1, v_2) \mid \Lambda\alpha.e \mid \texttt{pack}[\tau, v] \ \texttt{as} \ \exists\alpha.\tau' \\ \text{Expressions} & e & ::= \ x \mid v \mid e_1 + e_2 \mid e_1 \ e_2 \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e \mid e\ [\tau] \mid \texttt{unpack}\ \alpha, x \ = \ e_1 \ \texttt{in} \ e_2 \end{array}$$

Recall that $\pi_1 e$ and $\pi_2 e$ are expressions for projecting the first and second components of a pair respectively. We present the typing rules for the key expressions below (we omit rules for numbers, pairs and ordinary value functions – you've seen those before). Remember, Δ is a set of type variables ($\Delta = \alpha_1, \alpha_2, \ldots, \alpha_k$) and Γ is a function from value variables to their types ($\Gamma = x_1{:}\tau_1, x_2{:}\tau_2, \ldots, x_m{:}\tau_m$). Also remember that $\Delta \vdash \tau$ means that the free type variables of τ are a subset Δ. (Note also that this is sometimes called a "well-formedness" judgement for types and hence I decided to begin the rule names with WF.)

Typing Judgement Form I: $\boxed{\Delta \vdash \tau}$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ (\text{WFvar}) \qquad \frac{}{\Delta \vdash \texttt{int}}\ (\text{WFint})$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}\ (\text{WFarrow}) \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 * \tau_2}\ (\text{WFpair})$$

$$\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha.\tau}\ (\text{WFall}) \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \exists \alpha.\tau}\ (\text{WFexists})$$

Typing Judgement Form II: $\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau}\ (\text{Tforall})$$

$$\frac{\Delta \vdash \tau \qquad \Delta; \Gamma \vdash e : \forall \alpha.\tau'}{\Delta; \Gamma \vdash e\ [\tau] : \tau'[\tau/\alpha]}\ (\text{Ttyapp})$$

$$\frac{\Delta \vdash \tau \qquad \Delta \vdash \exists \alpha.\tau' \qquad \Delta; \Gamma \vdash e : \tau'[\tau/\alpha]}{\Delta; \Gamma \vdash \texttt{pack}[\tau, e]\ \texttt{as}\ \exists \alpha.\tau' : \exists \alpha.\tau'}\ (\text{Tpack})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \exists \alpha.\tau' \qquad \Delta, \alpha; \Gamma, x{:}\tau' \vdash e_2 : \tau}{\Delta; \Gamma \vdash \texttt{unpack}\ \alpha, x\ =\ e_1\ \texttt{in}\ e_2 : \tau}\ (\text{Tunpack})$$

And now, the operational rules:
Judgement form: $\boxed{e \longrightarrow e'}$

$$\frac{e \longrightarrow e'}{e\ [\tau] \longrightarrow e'\ [\tau]}\ (\text{OStyapp1})$$

$$\frac{}{(\Lambda\alpha.e)\ [\tau] \longrightarrow e[\tau/\alpha]}\ (\text{OStyapp2})$$

$$\frac{e \longrightarrow e'}{\texttt{pack}[\tau, e]\ \texttt{as}\ \exists \alpha.\tau' \longrightarrow \texttt{pack}[\tau, e']\ \texttt{as}\ \exists \alpha.\tau'}\ (\text{OSpack})$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{unpack}\ \alpha, x\ =\ e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{unpack}\ \alpha, x\ =\ e_1'\ \texttt{in}\ e_2}\ (\text{OSunpack1})$$

$$\frac{}{\texttt{unpack}\ \alpha, x\ =\ (\texttt{pack}[\tau, v]\ \texttt{as}\ \exists \alpha.\tau')\ \texttt{in}\ e_2 \longrightarrow e_2[\tau/\alpha][v/x]}\ (\text{OSunpack2})$$

Q. 1 [2 Points] Write down a typing derivation, excluding the derivations for judgements with the form $\Delta \vdash \tau$, for the following function, which creates a polymorphic function and then applies it to an integer argument:

```
(((Λα.
      λf:α → α.
         λx:α.f x) [int]) (λy:int.y+1)) 7
```

Q. 2 [2 Points] Here is an ML signature:

```
sig
  type complex
  val toComplex : int * int -> complex
  val fromComplex : complex -> int * int
  val add : complex * complex -> complex
end
```

Part (a) [1 point] Write the existential type that corresponds to the signature above.

Part (b) [1 point] Write down a lambda calculus expression that implements the signature given above. Your expression must have the existential type you gave for part (a) (you don't have to down write the complete derivation – just convince yourself you've done it correctly).

Q. 3 [2 Points] Write down the full statement of the canonical forms lemma for the language given above.

Q. 4 [4 Points] Prove the cases of the Progress lemma that involve existential types.

Progress: If $\cdot;\cdot \vdash e : \tau$ then either (a) e is a value v, or (b) $e \longrightarrow e'$ for some expression e'.

The proof is by inducntion on the structure of the typing derivation $\cdot;\cdot \vdash e : \tau$. You should assume the canonical forms lemma you stated in the previous question has been proven for you already. You must complete the following cases:

case:

$$\frac{\cdot \vdash \tau \qquad \cdot \vdash \exists\alpha.\tau' \qquad \cdot;\cdot \vdash e : \tau'[\tau/\alpha]}{\cdot;\cdot \vdash \texttt{pack}[\tau, e]\ \texttt{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}\ (\text{Tpack})$$

$$\vdots$$

case:

$$\frac{\cdot;\cdot \vdash e_1 : \exists\alpha.\tau' \qquad \alpha; x{:}\tau' \vdash e_2 : \tau}{\cdot;\cdot \vdash \texttt{unpack}\ \alpha, x\ =\ e_1\ \texttt{in}\ e_2 : \tau}\ (\text{Tunpack})$$

$$\vdots$$

Q. 5 [4 Points] Since the operational semantics of our language involves substitution for type variables, proving type preservation requires not only a value substitution but also a type substitution lemma. Here are statements of the two lemmas:

Type Substitution: If $\Delta, \alpha; \Gamma \vdash e : \tau$ and $\Delta \vdash \tau'$ then $\Delta; \Gamma[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$.

Value Substitution: If $\Delta; \Gamma, x : \tau' \vdash e : \tau$ and $\Delta; \Gamma \vdash v : \tau'$ then $\Delta; \Gamma \vdash e[v/x] : \tau$.

Assuming these two lemmas have already been proven for you, prove the cases for the type preservation lemma that involve universal polymorphic types.

Type Preservation: If $\cdot;\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot;\cdot \vdash e' : \tau$.

The proof is by induction on the operational derivation. Here are the cases you are required to complete:

case:

$$\frac{(p1)\ e_1 \longrightarrow e_1'}{e_1\ [\tau'] \longrightarrow e_1'\ [\tau']}\ \text{(OStyapp1)}$$

(1) $\cdot;\cdot \vdash e_1\ [\tau'] : \tau$ (By Assumption)

$\vdots$

Must prove: $\cdot;\cdot \vdash e_1'\ [\tau'] : \tau$

case:

$$\frac{}{(\Lambda\alpha.e_1)\ [\tau'] \longrightarrow e_1[\tau'/\alpha]}\ \text{(OStyapp2)}$$

(1) $\cdot;\cdot \vdash (\Lambda\alpha.e_1)\ [\tau'] : \tau$ (By Assumption)

$\vdots$

Must prove: $\cdot;\cdot \vdash e_1[\tau'/\alpha] : \tau$

Implementation

Q 6. [12 points] You will implement a type checker for the subset of the language studied in the previous sections that includes integers, functions and universal polymorphism. Here is the formal syntax of the language:

$$\begin{array}{llll} \text{Types} & \tau & ::= & \texttt{int} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall\alpha.\tau \\ \text{Expressions} & e & ::= & x \mid n \mid e_1 + e_2 \mid \lambda x{:}\tau.e \mid \Lambda\alpha.e \mid e_1\ e_2 \mid e\ [\tau] \end{array}$$

Your implementation should extend the file `polylam.sml`. The main challenge will be to implement these three key functions for the language:

```
(* implements type equality;
   return true if t1 and t2 are alpha-equivalent types *)
fun tyeq (t1:typ) (t2:typ) : bool = raise NotImplemented

(* implements the judgement D |- t *)
fun wellformedty (D:delta) (t:typ) : bool = raise NotImplemented

(* implements the judgement D;G |- e : t *)
fun typecheck (D:delta) (G:gamma) (e:exp) : typ = raise NotImplemented
```

The `tyeq` functions should implement the alpha-equivalence relation for types. In other words, `tyeq` should report that the types $\forall\alpha.\alpha \to \alpha$ and $\forall\beta.\beta \to \beta$ are equal. The `tyeq` function can then be used when implementing the type checking function `typecheck`. In particular, the implementation of a rule such as the rule for function application will use `tyeq` as a subroutine. As a reminder, here is the type application rule:

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{(Tapp)}$$

Notice that the rule says e_1 is expected to have type $\tau_1 \to \tau_2$ and e_2 is expected to have the type τ_1. Implicitly, since we use the same meta-variable (τ_1) in two different places in the rule, we are expecting the types in those two places to be "equal." "Equal" in this case means syntactically identical up to renaming of bound type variables (*i.e.,* we are implicitly checking for *alpha-equivalence* of types in these rules).

The precise rules for alpha-equivalence of types are presented in the appendix at the end of the assignment. The most interesting part of the equivalence relation involves the case for universal polymorphism since that case involves a bound type variable. The way to check if $\forall\alpha.\tau_1$ is equivalent to $\forall\beta.\tau_2$ is to pick some new variable α_1 (using the `freshvar` function provided to you) and to check if $\tau_1[\alpha_1/\alpha]$ is alpha-equivalent to $\tau_2[\alpha_1/\beta]$.

Checking alpha-equivalence of other types is fairly easy. For example, two type variables are only equal if they are the same variable. The integer type `int` is only equal to the integer type `int` – not to any other type. Note in particular that `int` is not alpha-equivalent to a type variable α. (We are checking for alpha-equality, which is quite different from doing *unification*, which is a technique used in in our *type inference* algorithm.) The type $\tau_1 \to \tau_2$ is alpha-equivalent to $\tau_1' \to \tau_2'$ if and only if τ_1 is alpha-equivalent to τ_1' and if τ_2 is alpha-equivalent to τ_2'.

In addition to alpha-equivalence, there is one other slight twist in our implementation. That is that we will implement the context Γ as a list of variable-type pairs. To add to Γ (such as in the rule for typing functions), we will put a new variable on the front of the list (use the `update` function provided to you). To find the type associated with a variable, scan the list from front to back, stopping when you find the first occurrence of the variable in question (use the `lookup` function provided to you). If you do this, you can actually add the same variable to the context twice and the lookup function will find the last one added, which will result in the correct behavior.[1]

A complete set of type checking rules is provided for your reference in the appendix, starting on the following page.

[1]Treating contexts as ordered lists like this is an alternative to explicitly alpha-varying bound value variables to make them different from the value variables that already appear in the context, as the typing rules technically require.

Appendix: Type Checking Rules for Implementation

$\boxed{\tau_1 \equiv \tau_2}$ (Alpha-equivalence relation used implicitly)

$$\frac{}{\alpha \equiv \alpha}\ \text{(EQvar)} \qquad \frac{}{\texttt{int} \equiv \texttt{int}}\ \text{(EQint)}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}\ \text{(EQarrow)}$$

$$\frac{\alpha_1 \notin Var(\tau) \cup Var(\tau') \qquad \tau[\alpha_1/\alpha] \equiv \tau'[\alpha_1/\beta]}{\forall\alpha.\tau \equiv \forall\beta.\tau'}\ \text{(EQall)}$$

$\boxed{\Delta \vdash \tau}$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ \text{(WFvar)} \qquad \frac{}{\Delta \vdash \texttt{int}}\ \text{(WFint)}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}\ \text{(WFarrow)}$$

$$\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall\alpha.\tau}\ \text{(WFall)}$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}\ \text{(Tvar)} \qquad \frac{}{\Delta; \Gamma \vdash n : \texttt{int}}\ \text{(Tint)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \texttt{int} \qquad \Delta; \Gamma \vdash e_2 : \texttt{int}}{\Delta; \Gamma \vdash e_1 + e_2 : \texttt{int}}\ \text{(Tadd)}$$

$$\frac{\Delta \vdash \tau_1 \qquad x \notin Dom(\Gamma) \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e_1 : \tau_2}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \text{(Tfun)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}\ \text{(Tapp)}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \text{(Tforall)}$$

$$\frac{\Delta \vdash \tau \qquad \Delta; \Gamma \vdash e : \forall\alpha.\tau'}{\Delta; \Gamma \vdash e\ [\tau] : \tau'[\tau/\alpha]}\ \text{(Ttyapp)}$$